FILE MANAGEMENT AND RELATED TOPICS

Robert M. Graham

MAC Technical Memorandum 12

September 1970

(Formerly Programming Linguistics Group
Memo No. 6, 12 June 1970)

This informal document has been published to make the research results, though perhaps tentative, quickly available to a limited audience.

Massachusetts Institute of Technology

PROJECT MAC

545 Main Street

Cambridge 02139

5d

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP None

3. REPORT TITLE

File Management and Related Topics

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Memorandum

5. AUTHOR(S) (Last name, first name, initial)

Graham, Robert M.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
September 1970	53	0

8a. CONTRACT OR GRANT NO. Nonr-4102(01) b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) TM-12 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D.C.

13. ABSTRACT

~~This~~ paper traces the evolution of a segment based file system. The final system is typical of the virtual memory systems found in large general purpose time-sharing systems. The contents of the file system is a collection of symbolically named segments organized in a hierarchial structure. The user directly references segments in the file system. All movement of information between the different levels of physical memory is done automatically by the system using paging. Complete privacy of user information is guaranteed, although controlled sharing is possible. The system includes file backup facilities to protect users from information loss due to system failure.

14. KEY WORDS

File System Virtual Memory Paging Segmentation Time-Sharing Systems
Input-Output Memory Management Multi-level File Storage File Sharing
File Protection

DD FORM 1 NOV 68 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

ACKNOWLEDGMENT

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

Note: Second International Seminar on Advanced Programming Systems, The Hebrew University of Jerusalem and Iltam Corp., Jerusalem, Israel, August 1969.

FILE MANAGEMENT AND RELATED TOPICS

Technical Memorandum 12

(Formerly Programming Linguistics Group
Memo No. 6, 12 June 1970)

Robert M. Graham

September 1970

PROJECT MAC

Massachusetts Institute of Technology

Cambridge

Massachusetts 02139

FILE MANAGEMENT AND RELATED TOPICS

Robert M. Graham
Massachusetts Institute of Technology

1. Introduction

The subject of these notes is file management. We will develop the problems of file management within the environment of a large information and computing service, often called a computer utility or general purpose time-sharing system. We do this for two reasons. First, this environment imposes the most severe constraints. Other environments are obtained by relaxing these constraints. Secondly, large information and computing services will become more and more prevalent in the years to come.

Let us first look briefly at those objectives of an information and computing service which are significant to this discussion.

a. Continuous service

The system must normally run 24 hours a day, seven days a week. This implies a high degree of reliability. It also implies that maintenance of the system must be done on-line.

b. Multi-user

The system must be able to service a large number of users working on diverse applications. This implies some sort of resource management in order to parcel out the time and storage facilities to this large community of users. In addition, since users may very well be competitive and even non-competitive users may storage sensitive data in the system, protection of user's privacy must be guaranteed.

c. Permanent information storage

The system must be able to store permanently as much information as the user desires and be able to retrieve it undamaged at some future time.

d. On-line, conversational

The system must be able to provide on-line, conversational facilities to a large number of the users at any given time. This implies some minimum response time for trivial requests. In addition, it implies the need to be

able to simultaneously process input and output from many terminals. Finally, it implies some kind of multi-programming is necessary.

e. Facilitate cooperative efforts

The system should facilitate cooperative efforts within groups of users. This implies that sharing is necessary and further, that sharing must be controlled. In addition, it implies communication among users via the system itself.

2. The Function of an Operating System

Most operating systems have two major functions, the management of resources and provision of an interface between the user and the physical hardware. Let us look briefly at both of these aspects of an operating system.

2.1. Resource Management

The resource management aspect of an operating system is based on the observation that there are a number of resources in the system which need to be managed. These are the processors, memory (core, drum, disk,...), I/O devices, and information (programs, data). The naive view taken in the early days of computing was that the user should manage all of these resources. The user approached the bare machine with whatever cards and tapes he needed and the entire set of resources were his to do with as he pleased. He was also stuck with the job of programming their management. Today we recognize that this is not practical as a method of operation. Most operating systems today take over a large portion of the management of these resources for the user. Within the framework of resource management our view of file management is that it includes the entire storage management, including the management of core memory as well as file memory and information management. It is important in thinking about resource management to keep separate the two aspects: mechanics and policy. There is also great advantage in separating these two aspects in any implementation. We have the goal of implementing the system so that the system administrator is allowed as much freedom as possible in choosing his allocation policy. By implementing policy decision in a module which is separate from the mechanics of allocation, which is not easily changeable, policy can be easily changed.

2.2. The System As an Interface

In this section we view the system as a program which implements a mapping between the user's conception of the world and reality. There are, in fact, several levels of the user's view of the world, depending upon his experience and his intent. One can easily distinguish the levels: a casual user of an application subsystem, an ordinary programmer using PL/I, a subsystem implementer, and finally, a system programmer. A user may have quite different conceptions of the system at the various levels. As an example of the system implementing a mapping between the user's conception of the world and reality, we will see a file system in which the user refers to files by symbolic names and has the illusion that memory is homogenous. The reality is that memory is not homogenous. It is composed of disks, drums, core, etc. References to files can not be made symbolically but must be made by using device number, track number, etc.

We also note that resource management may hold at each of the different levels. Again the view is different depending upon the level. For example, in the file system the user is able to create and destroy files. He has a limit on the total length of all files which are charged to him. The limit may be expressed either as the maximum number of words or as the number of dollars which he may spend. He generally has no need to be concerned with where the files are actually stored. The system does have to be concerned. It maintains a record of free storage for the various devices and must find a place to put the file when the user creates it. The system has an absolute limit on the total number of words of information which may be stored on any particular device.

3. A Model System

In this section we give an overview description of a model information and computing service. This is not a real system nor even a simplification of a real system. Rather, it is an amalgam of the author's knowledge and experience in this area, sufficiently simplified so that the important concepts involved may be clearly exhibited. It is very similar to several existing systems, principally Multics and TSS/360.

3.1. The User's Conception of the System

We first look at the user's conception of the system. When a user is active he owns one or more processes. A process is, somewhat imprecisely, a program in execution. Each process executes on a processor which belongs exclusively to it as long as the process exists. A process may stop itself to wait for the occurence of some event. At some later time, when the event occurs, it starts executing again. A process can create and destroy other processes. Processes may communicate with one another, i.e., send signals and messages in a stylized, formal way. Since a process is a program in execution and runs until it stops itself, the process may be visualized as some sort of virtual computer. We are interested in two aspects of the virtual computer: the address space which defines the set of information which the process may reference, and the processor that the process owns and executes on, which we call the pseudo-processor. The pseudo-processor is implemented by the traffic controller and the address space is implemented by the storage manager.

The user views the address space as a hierarchy of files. The user may freely create new files and place them anywhere in the hierarchy. He may add additional levels to the hierarchy. He may delete files which exist in the hierarchy. Aside from some simple space allocation, he views memory as being essentially unlimited in size. His references are direct, symbolic references. These references are device and location independent and do not require any open, read, or write calls. The address space of his process is shared with other processes so that immediate sharing of information is possible. Finally, the memory has automatic backup with the ability for the user to retrieve any files which become damaged.

A user's process executes in a particular environment. He sees input as a source and output as a sink. References to input and output are symbolic and the reading and writing are device independent. Symbolically referenced input or output must be attached to a physical device. However, the attachment is dynamic and may be modified during execution. Each process is provided with a push-down stack for use in calling other procedures and for data of the type known as automatic in PL/I. Each process is also supplied with an area for use by procedures for their private data, data which is called internal static in PL/I.

The user controls the system using a command language which includes a macro-command facility. The user has mobility between interactive and absentee user type service. Movement between the two types of service may be done upon command and does not require reprogramming. All programs, including the system itself, commands, and user programs are written on the same base, i.e., they use the same language processors and they observe the same conventions, such as, procedure calling sequences and data formats. All programs, including the system, also execute in the same address space of the pseudo-processor.

3.2. Reality

Let us look briefly at the reality of this model system. The hardware has either a single or a very small number of processors which must be shared by all the processes running in the system. It is the task of the traffic controller to multiplex processes among the processors. Every process in the system is in one of three states: running, ready, or blocked. A process is running if it is actually executing on a processor. A process is ready if it could run if a processor were available although none is available at the present time. The user is unaware of this state of a process. It is indistinguishable from the running state as far as he knows. A process is blocked when it is waiting for some event to occur before it may proceed. The user may be aware of a process in this state if the event is one for which he is waiting. On the other hand, the event may be in response to a system imposed wait, such as an input/output completion. In this case, the user is unaware of the blocked state of his process.

The storage manager is responsible for implementing the address space. The address space of the real processor in the system is a segment address space. A hardware interpretable address is a pair of integers consisting of a segment number and a location within that segment. In order for a program to execute a reference to data, it must have an address of this form. Hence, the job of the storage manager is to map files, which may in fact be stored on disk, into the hardware address space in order that the process may reference them. In addition to transforming a symbolic reference into a machine address, the system must be concerned with moving information from the disk to the core memory in order for it to be referenceable.

4. Management of the Hierarchy

The user's conception of memory is a hierarchy of files. This hierarchy is a tree structure of files. The management of this structure is the topic of this section. It is based on the concept of segment. A segment is a symbollically named collection of information which is a basic unit of organization used by both the user and the system. A segment is a contiguous set of words (or bytes), e.g., a table, a procedure, etc. A segment may have internal structure which is interpreted by the user. However, the system assumes no internal structure other than an array of words. An example of a segment which is familiar to all of us is a book in a library.

4.1. A Directory

A hierarchy will be composed of segments. Segments and files are identical. Segment seems more appropriate to us, however the term file will still be used, especially when the referenced information is physically stored in file memory (e.g., disk). Certain information is needed for each segment. The following items are obvious now (more will be added later):

a. name

b. length

c. location

(Note: In this and the next few sections we will assume that core memory is large enough to store, simultaneously, all the segments which exist in the hierarchy at any given time. This will allow us to simplify the description of the location and also permit us to state the memory allocation problem in its simplest terms. We will later remove this assumption and see the implications of the removal.) We will collect together all of this information about all of the segments into a directory which contains one entry for each segment. Further, no segment shall exist without a corresponding entry in the directory. A directory is like a card catalogue in the library. In fact, in some systems a directory is called a catalogue.

Once we have a directory there are certain operations that need to be done to the entries in the directory. These operations, collectively, we will call directory management. The manipulations of the directory required are:

a. Add an entry to the directory, i.e., create a segment.

b. Remove an entry from the directory, i.e., delete a segment.

c. Modify a directory entry, i.e., change the name of a segment or change the length of a segment.

d. Copy information from a directory entry.

4.2. The Hierarchy

Now we have all the information about the segments assembled into a single directory. Are there any disadvantages of such an arrangement? The answer is yes. Two of the principal difficulties are name conflicts and the size of the directory. The problem of name conflicts becomes very severe in a system with a large number of users. Each user has his own files and it may be very difficult, certainly irritating, for the user to always have to scan a large list of names in order to find one which is not already in use every time he creates a new file. The size of the directory, which will be very large in the case of a large system, certainly complicates the searching process in addition to slowing it down.

There is another disadvantage which is not quite so obvious: a single directory does not contain any content related structure. Content related structure is very convenient and desirable. We are all aware of the content related structure of a library's stacks. In a large library with a large number of books we find the stacks divided into sections such as, History, Science, and Philosophy. History is further subdivided into the subsections American History, French History, etc. The American History section might be again subdivided into various subsections pertaining to the periods in American history, such as Pre-Revolutionary, Reconstruction Era, etc. The convenience of such an arrangement is very significant, particularly when you know the subject but don't know the title. You need to scan all of the entries on that subject. If no content related structure were present one would be faced with the task of scanning the entire card catalogue in the library, a monumental, if not impossible, task.

This type of structure is called a tree or hierarchy. There are many examples of this type of structure both in life and in the computing field (e.g., PL/I structures which are all hierarchies). We implement such a hierarchy by having a number of directories which are related to each other

in that certain directories contain entries pointing to other directories rather than to data segments. In order to do this we need an additional piece of information in the directory entry: type information indicating whether the entry describes another directory or data segment. Figure 1 shows part of a typical hierarchy of segments for a large information and computing system. Only the names of segments and directories in a given directory need be unique. The same name may be repeated over and over again as long as each time it is in a different directory. This resolves the problem of name conflicts. On the other hand, how does one reference a segment in such a structure? There is one method of reference which allows completely unambiguous references. It consists of the concatenation, separated by periods, of all the names of all of the parent directories of the segment being referenced. We call such a reference an absolute tree name. For example, in figure 1 the segment marked with * has the absolute tree name ROOT.E.A.D, while the segment marked with ** has the absolute tree name ROOT.E.A.C.R.A.

Another useful feature in such a structure is a cross reference, which we call a link. This is an entry which points to another entry rather than to an actual directory or data segment. This is a form of indirection. In figure 1, ROOT.E.A.Q is a link. It names the same segment as does ROOT.E.A.C.R.C. In order to implement a link we expand the type information to indicate non-directory, directory, or link. For the purpose of easy management all other information about this segment will be kept in the master entry. Hence, in the link entry the length is not used and the location, rather than being the physical location of the segment, will be the absolute tree name of the master entry for the segment.

5. Physical Storage Management

We will continue to assume that we have a large enough core memory to hold all of the segments in the hierarchy at any given time. There is still a problem since we are not assuming that memory is large enough so that we never need to reclaim any of the space occupied by segments which get deleted. This is the simplest situation in which to frame the basic problem of storage management or allocation. We will later address the actual situation which is that of a small core memory backed up by drum, disk, and tape, i.e., a

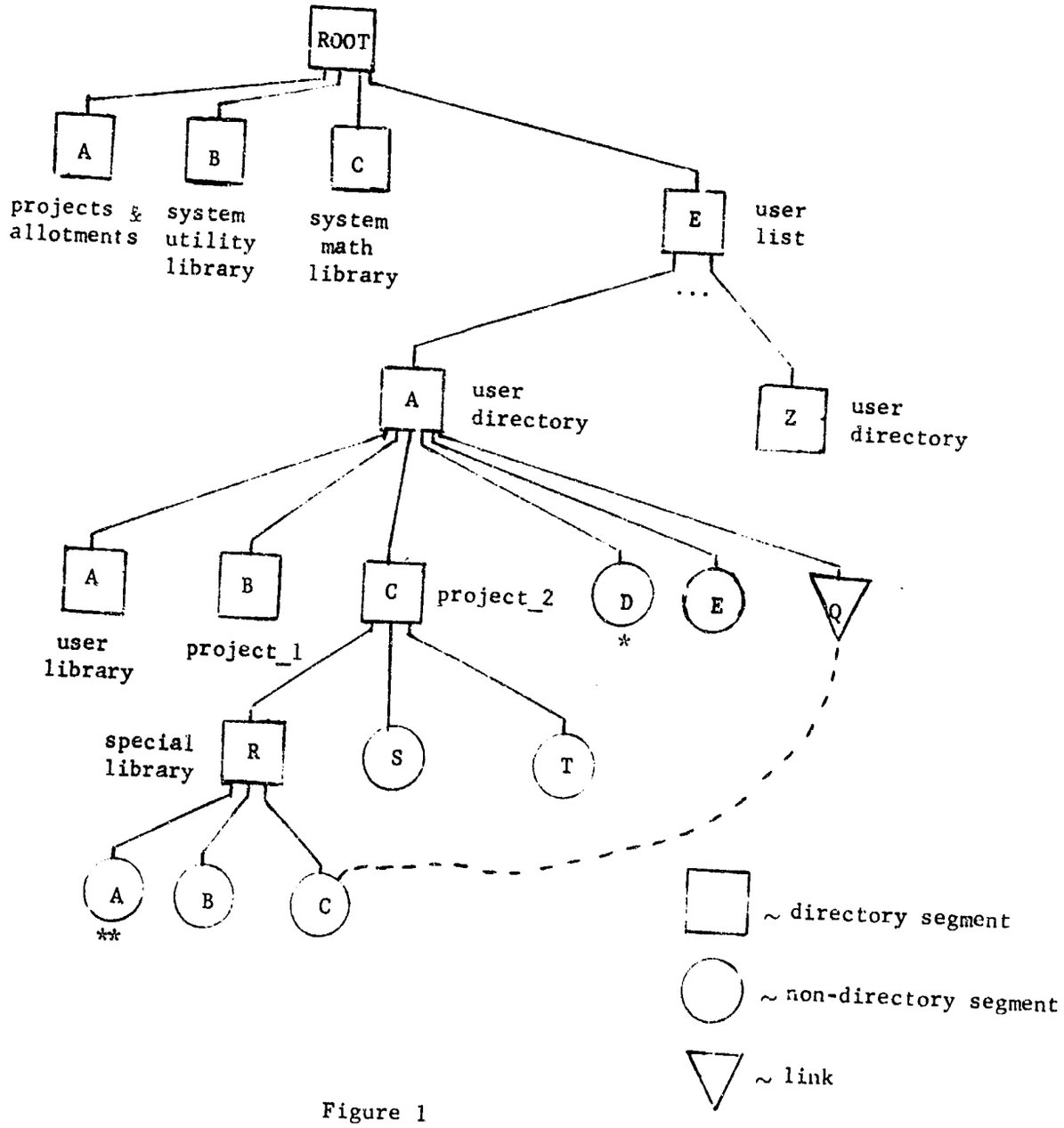
ROOT
A
B
C
E
projects & allotments
system utility library
system math library
user list
...
A
user directory
Z
user directory
A
B
C
project_2
D
E
Q
user library
project_1
*
special library
R
S
T
A
B
C
**
~ directory segment
~ non-directory segment
~ link

Figure 1

non-homogeneous, multi-level memory.

An address space is a set of reference labels, such as tuples, integers, etc. A segment is a linear array of words (or bytes) with a symbolic name, S. The address of a word in S is the pair (S,W), where W is the offset of the word within the segment S. (S,W) is a reference label in the user's address space (which we will call the hierarchy name space). Since a major task of the file system is the mapping between the user's address space and the hardware address space, it seems reasonable that the hardware address space reflects, as closely as possible, the structure of the user's address space. This is part of the motivation for the hardware address space being segment oriented, with an address composed of a segment number and a word number. We will see later other motivations for the segment addressing hardware.

Before we turn to an examination of the actual hardware addressing, let us look briefly at how the memory will be managed. There are two major techniques for memory management, contiguous allocation and block allocation. We will discuss each in turn.

5.1. Contiguous Address Allocation

When we create a new segment of length N we need to find a space of N contiguous words, starting say at location α. In referring to a word in the segment we use the pair (α,W). The absolute address, A, of word W in the segment is $A=\alpha+W$ (see figure 2). In contiguous address allocation, as with all methods of allocation, we need a table or list of the unused, or free, locations in memory.

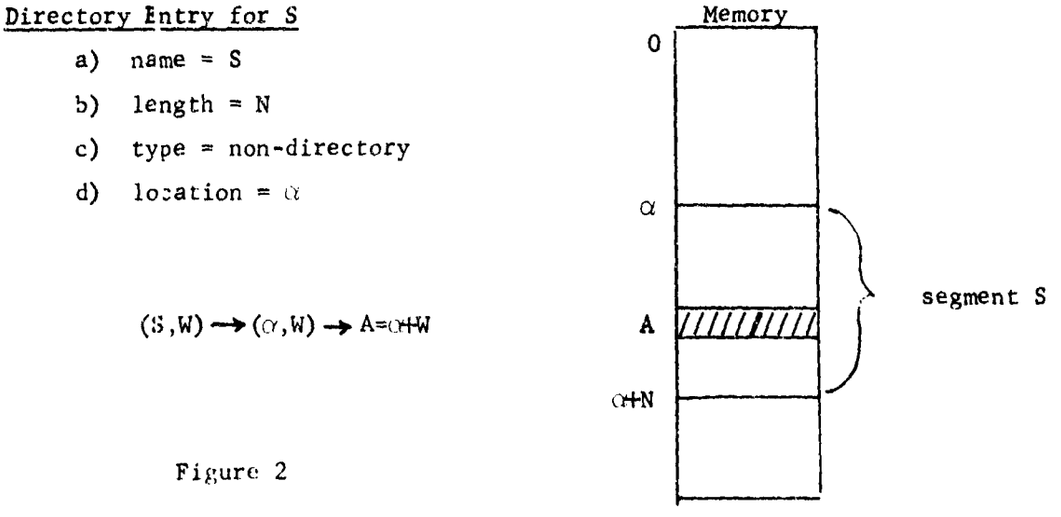

Figure 2

Our problem arises when N is larger than the length of any contiguous block of free space, even though the total amount of free space is larger than N (this being part of our original assumption). In order to get out of this dilemma we need to move other segments in memory in such a way as to compact the unoccupied memory in order to obtain enough contiguous space for our new segment. All the users of segments which had to be moved must be told about this since the ID of each of these segments changed. All of their address references will now need to be recomputed. Since memory allocation is dynamic all references to segments must either remain continuously unbound, being bound each time a reference is made or the references must be unbindable so that they can be unbound and rebound whenever memory is compacted.

Let us be more precise about the particular algorithm that is being used here. We will compact all of the segments at the lower end of memory. Whenever a new segment is created we will assign it to the free space just above the top of the used part of memory. Whenever a segment is deleted we will just let it go, but remember that the space which it occupied is now empty. When the free area is small enough that a segment about to be created will not fit we must stop and compact all of the segments in memory. It is interesting to look at the amount of time which will be spent in compacting.

Let M be the size of the memory and F be the fraction of memory used, then (1-F)M is the amount of free space. Suppose further that K is the average number of references to each of the words in a segment before the segment is deleted. Since there is at most one reference to memory per time unit this means that every K time units one word will be deleted. Assuming equilibrium one word will be added every K units. Hence, the free area will be exhausted in (1-F)MK units of time. The time required to move the information in the remaining segments is about 2MF (assuming two references to move one word). The fraction of time spent in compacting is then

$$\frac{\text{compacting time}}{\text{compacting time+time to fill free storage}} = \frac{2MF}{2MF+(1-F)KM} = \frac{F}{F+(1-F)K/2}$$

In a time-sharing system where there is heavy use of functions such as editing, a reasonable figure for K is around 20. With K=20 and F=50%, we see that the function of time spent compacting is 9%. With K=20 and F=75%,

the time spent compacting is 25%. The implication of this is that to achieve a reasonable compaction time of less than 10% we have to give up over half of the memory.

5.2. Block Allocation

In this scheme we divide memory into blocks of K words. A segment of length N is then divided into $B = \left[\frac{N-1}{K}\right]+1$ pages. Each page is stored in a block, so the segment is stored in B blocks of memory. However, these blocks need not be contiguous. In order to achieve this non-contiguity we need a page map which contains the location of all the blocks in the segment. A reference to a word in the segment is again a pair (α,W). α is the location of the page map rather than the first word of the segment and W is the offset of the word in the segment. To find the absolute address of the word in memory we must find two integers I and J such that I is the base address of the correct block and J is the word number within the block. The absolute address is then, A = I+J. The page number, in the segment, is $P = \left[\frac{W}{K}\right]$. Then I is the contents of α+P, i.e., the page table entry for page P. Finally, J = W-PK, i.e., J = W(Modulo K). See figure 3.

Directory Entry for S

a) S
b) N
c) non-directory
d) α

$(S,W) \rightarrow (\alpha,W) \rightarrow (I,J) \rightarrow A=I+J$

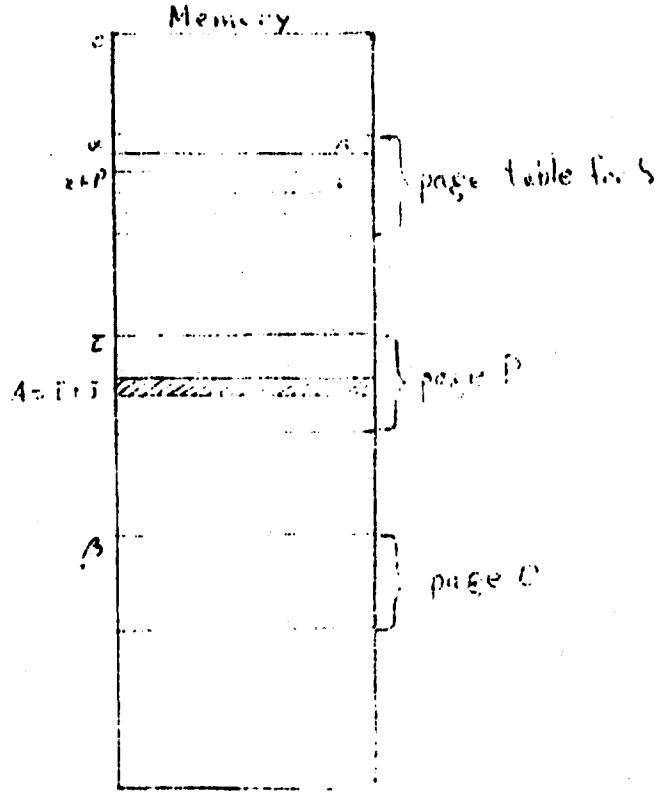

Figure 3

Two memory references are required to access a word using this scheme. The first memory reference obtains I from the page table, the second memory reference obtains the word of data which is being referenced. There are two disadvantages to this scheme. The first is breakage. On the average $\frac{K}{2}$ words are unused in the last block of each segment (in addition 1 extra block is required as a page map for each segment). The second disadvantage is the additional memory reference required to access each word.

5.3. Comparison of the Two Methods

The following chart is a summary of the two methods compared for various properties.

Allocation Type	Contiguous	Block
Reference time (to one word)	1	2
Data movement	Compacting required	No movement needed
Waste space	Up to 50% to reduce compacting time	Breakage + Page Map
User program reference	Rebinding required on movement of segment	No rebinding needed

We note that with modern associate memory techniques the extra reference required per word for the block allocation scheme can be significantly reduced. In fact, hardware is available on a number of computers which make the block allocation scheme much more attractive than the contiguous allocation scheme. We now look at an example of hardware.

5.4. Segment and Paging Hardware

In this section we describe some actual segment and paging hardware. It is the segment and paging hardware found on the GE645 computer. Other computers, notably the IBM 360/67 have hardware which is practically identical to that which is described here. A hardware address is a pair of 18 bit integers. The first integer is a segment number, the second integer is a word number within the segment. This allows 2^{18} segments and 2^{18} words in each segment. The segment number is used as an index in a descriptor table. The word number is further split into two fields. The high order 8 bits is the page

number and is used as an index in a page table. The low order 10 bits are the word number in the page specified by the first 8 bits. Figure 4 diagrams these relationships.

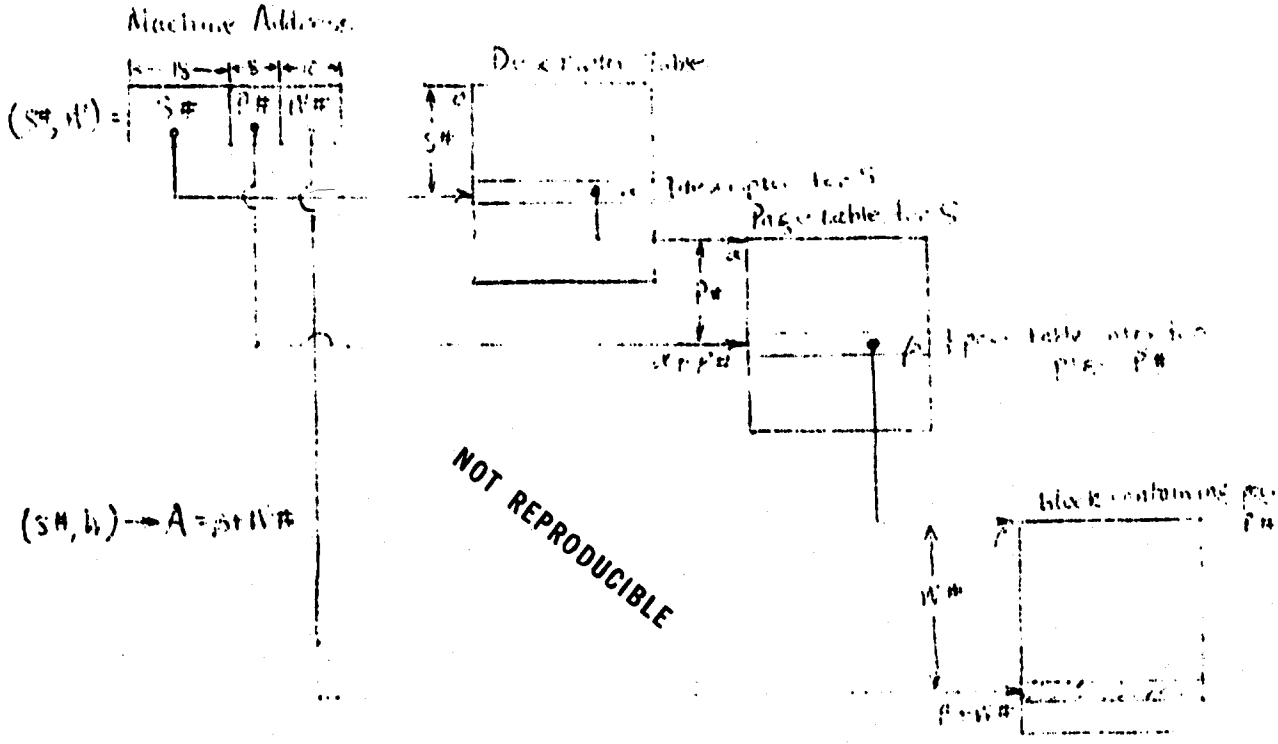

Figure 4

The entry in the descriptor table contains the address of the beginning of the page table. The entry in the page table contains the address of the beginning of the block in which the page is stored. The absolute address of the referenced word is then A = β+W#. Although the descriptor table and the page table are required to be stored in core memory the mapping is applied automatically by the hardware on each reference. The user and the system, in general, may not refer to information in any other way, i.e., every reference to memory is a segment-page reference. Thus the hardware implements an address mapping from a segment space consisting of segment numbers and locations into a linear address space consisting of a set of contiguous locations addressed by a single integer which is an absolute address. Note

that when using this hardware the location stored in the directory entry is the segment number, S#, rather than the base of the page table, α.

5.5. Core and Segment Management

We postulate two modules, core management and segment management. They are responsible, respectively, for the management of core memory (i.e., the linear address space) and the managment of segment numbers (i.e., the segment address space).

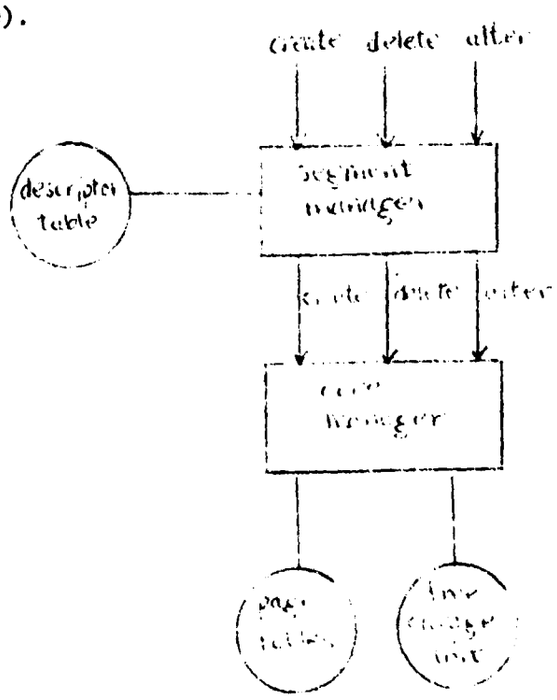

Figure 5

The core manager has three entries: create, delete, and alter. The create entry will find a sufficient number of blocks for a segment of length N and a page table to go along with it. It builds a page table with entries pointing to the blocks assigned to the segment and returns the address of the page table. The delete entry will release to free storage the space pointed to by the entries in a page table whose address is its argument. The alter entry will either obtain or release space and alter the page table to reflect

the change in length of a segment requested by the entry.

The segment manager also has three entries: create, delete, and alter. However, at this level we are working with segment numbers rather than absolute addresses. The create entry will call the core manager to obtain space and a page table for the segment to be created. The next unused segment number will be assigned to the segment. A descriptor for the segment will be put in the descriptor table. It contains the absolute address of the page table which is returned by the core manager. The segment number of the newly created segment is returned by the segment manager to its caller. The delete and alter entries take as arguments segment numbers which are transformed by the segment manager, via the descriptor segment contents, into the absolute address of a page table which is then used as an argument to call the core manager to have the actual space released or obtained.

6. Mapping Between the Hierarchy Name Space and the Segment Address Space

We now discuss the mapping between the hierarchy name space of tree names and the segment address space of segment numbers. Stated in another way the problem we are discussing here is: given the absolute tree name of a segment, such as ROOT.X.Q.Z, obtain the segment number of that segment. The problem conceptually is quite simple, in the sense that the segment number for the named segment appears in the directory entry for that segment. Hence, all we need to do is find the appropriate directory entry.

Let us review briefly the functions which the directory manager module can perform for us. The directory manager has a number of entries. Each entry always has one argument which is the segment number of a directory upon which the function is to be performed. The functions are:

a. Create, which adds a new entry to the directory.
b. Delete, which removes an existing entry from the directory.
c. Alter, which modifies the contents of an entry in the directory (such as changing the name or the length of the segment).
d. Copy, which copies the information from the directory entry.
e. List, which makes a list of all of the entries in the directory.

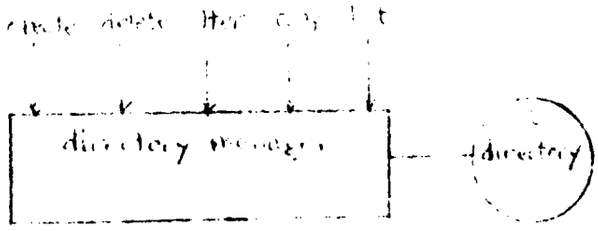

Figure 6

6.1. Directory Search

Each directory is in fact a separate segment. In order to refer to any information the hardware processor must have a segment number. Hence, in order for any program, such as the directory manager, to refer to a directory it must have the segment number of the segment containing the directory. This recursive situation, i.e., the directory manager being the only program which can find a segment number needing a segment number in order to find one, is broken by having one fixed segment number in the system, namely, the segment number of the root directory (ROOT in figure 1).

The algorithm then is as follows: We use an additional module called search control which calls the copy entry of the directory manager. See figure 7. Given the name ROOT.X.Q.Z, search control calls the directory manager three times in succession. On the first call the arguments are the segment number of the root directory, which is known, and the name X. The directory manager returns the segment number of X. The directory manager is then called with the arguments, the segment number of X and the name Q. The segment number of Q is returned. The final call to the segment manager has the arguments, the segment number of Q and the name Z. The segment number Z is returned. Search control is then able to return the segment number corresponding to the segment name with tree name ROOT.X.Q.Z.

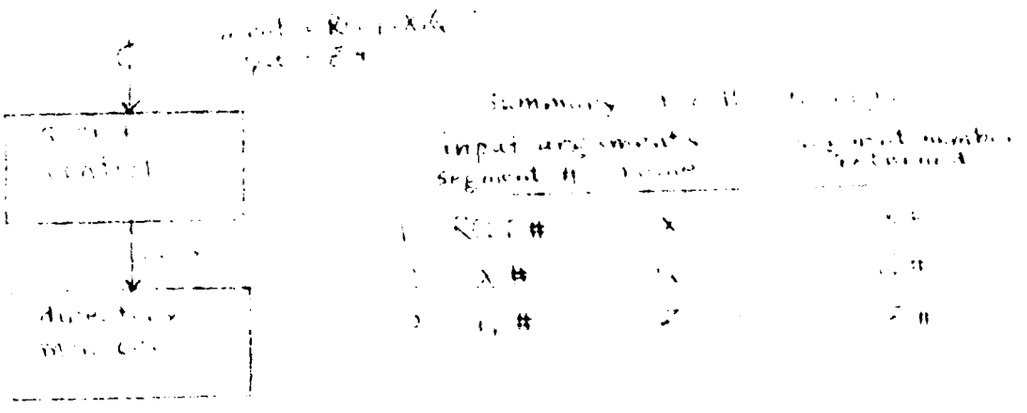

Figure 7

6.2. Linking of Procedures and Data

Any system which permits separately compiled or assembled procedure segments to reference each other symbolically must provide a facility for linking these separately compiled segments together, either before execution begins or during execution. In order to achieve this linking at some time other than compilation, the compiled procedure needs an appendage which contains information regarding the symbolic names of the segments which are referenced externally. If a reference is made by a procedure to another procedure segment named, ROOT.LIB.TRIG, this symbolic name must be contained somewhere in the output of the compiler.

We have been speaking so far about symbolic references to segments which ultimately result in a segment number. Most programmers are familiar with the concept of symbolic reference to locations within other segments. This feature is provided in most loaders today. We see that basically the problem is no different for this case then for the case where just the name of a segment is symbolic. Our appendage produced by the compiler must also contain symbolic information for any external references to locations within other segments. In the previous example, we probably would be referring to a specific subroutine within the segment ROOT.LIB.TRIG. Suppose it is the sine subroutine and the symbolic name of the entry point within the segment is SIN. The complete

source language reference might be ROOT.LIB.TRIG$SIN. This implies that the appendage produced by the compiler which compiles the segment ROOT.LIB.TRIG must contain a definition of the symbol SIN. Supposing that the entry point were at relative location 42, the appendage would contain the pair (SIN, 42) defining the symbolic location SIN. This concept of an appendage containing the necessary information for symbolic references to external segments and the definitions of symbols within the segment which may be referenced externally, should not be new or strange to anyone familiar with the loaders found in most modern operating systems.

As mentioned earlier there are two times at which linking can logically take place. One is before execution and the other is during execution. The typical loader in most systems does pre-execution linking. All of the segments which are needed to execute the program are linked together before execution begins. In many systems this is called loading. In earlier days the loader usually lead directly to execution. Recognition that the function of linking a separate process which need not lead to immediate execution can be seen in the terminology used in OS/360 where the loader is now called the link editor.

A loader operating in the environment which we have been discussing would build a table containing each of the symbolic references. Search control would be called once for each name in the table to obtain the segment number of the segment. This segment number would be added to the table entry for the symbol. Using the known conventions for the location of the appendage to the segment, all of the symbolic locations within segments would be looked up in the appropriate segment's appendage. These definitions would also be entered into the symbol table. Finally, all symbolic references would be replaced by the appropriate segment pointer, i.e., by a pair (S#,W).

Pre-execution linking has a number of disadvantages which have motivated the dynamic linking facilities which are available in the new, large information and computing systems like Multics and TSS/360. The major disadvantages of pre-execution linking are:

1. Many segments which are never used may have to be linked together in a large complicated program complex.
2. In the system of the type we are discussing it is difficult, if not impossible, with the tempo of interactions and the continuous

progression from command to command to determine when execution begins and when it ends. In fact, it is impossible to subdivide an interactive conversation program into loading and execution phases in any meaningful way.

3. Names of segments which are to be referenced by a procedure are often not known until after execution begins. In fact, they may be input data to the program which is executing. Consider, for example, an edit program or a compiler. These programs do not know the names of the segments they are going to refer to until they begin execution.

Dynamic linking (during execution) is not conceptually more difficult than pre-execution linking. A procedure which is going to be dynamically linked is fixed by the compiler so that the first time it attempts to make an external reference, a fault will occur. The appendage which includes the symbolic name of the external reference is included along with the procedure at execution time. The fault handler for the fault which occurs when the first reference is made, establishes the link at that time. Hence, this fault handler is called the linker and the fault is called a link-fault. The linker, using information given to it by the hardware when the fault occurred, is able to work its way back to the procedure which caused the fault and find the symbolic information necessary to define the external reference. Using this information, i.e., the tree name of the segment being referred to, the linker calls search control to obtain the segment number of the referenced segment. The linker than replaces the faulting reference with the appropriate segment pointer and restores the machine conditions. Execution then continues at the point the fault occurred.

6.3. Local Names

It is highly undesirable to require the user to always use absolute tree names in writing his source language program. The user is usually working in a well defined context within which local names can easily be interpreted. For example, the names of any referenced segments which are not in his user directory should be interpreted as system library procedure names.

Let us modify the linker so that it is able to deal with local names. In order to achieve this we need to interpose between the linker and search control a module which expands the local name into an absolute tree name.

This module uses as its principal data base a set of context rules which are used to transform local names into absolute tree names.

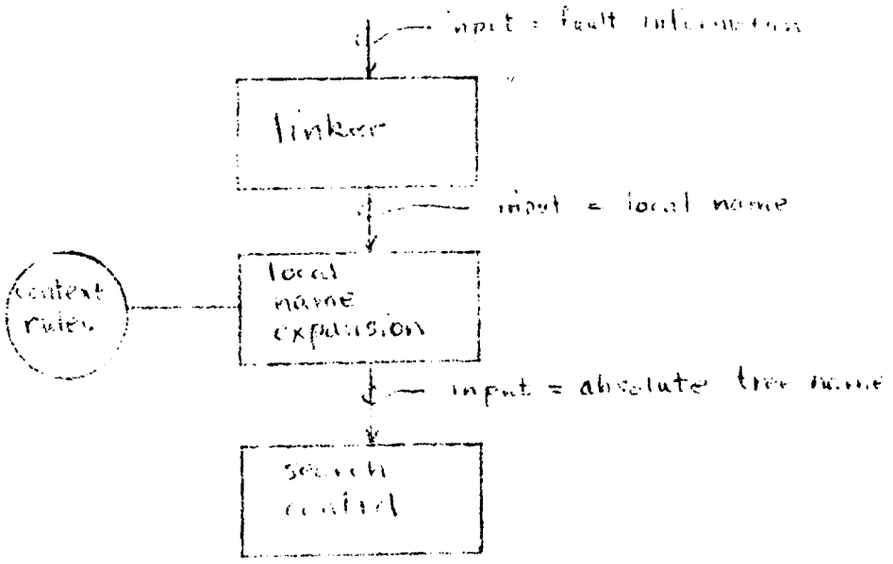

Figure 8

The simplest kind of context rule is simply a directory name in which to look for the local name. Thus a set of context rules would be a list of directory names. The mapping in this case is quite simple. The first directory name is prefixed to the local name, making an absolute tree name. This tree name is passed to search control in an attempt to find a segment with that name. If search control is not successful, the next directory name on the list is prefixed to the local name. Search control is again called to see if a segment can be found with the new name. This process continues until the list is exhausted. If no segment was found it is assumed not to exist and appropriate error action is taken. An example list of context rules begins with the name of the working directory. This is a directory declared to the system by the user to be his home directory. It is usually his user directory or one of his project directories. The second directory on the list would be the system utility library directory. Succeeding entries on the list might be other system library directories. The user must be able to control

this list of context rules; reorder them, add to them, or delete from them. For example, the user may wish to include several of his own private library directories between the working directory and the system library directories on the list. The definition of context can be made more flexible by allowing conditional rules, iteration, etc. In other words, the context rules can be expressed as a program in a simple language.

6.4. Local Associative Memories

In any complex of programs it is highly probable that more than one procedure will refer to a given segment. Each such reference initially requires the invocation of the linker, an application of the context rules, and a number of directory searches. It is possible to avoid these additional directory searches and context rule applications if the linker maintains a table whose entries consist of local names and their corresponding segment pointers. The linker then searches this table for a local name before calling local name expansion. If the local name exists in this table there is no need to make any further calls. If the local name can not be found in the table its segment pointer is obtained by calling local name expansion. The local name along with its segment number are then entered in the table. We call this table a local associative memory since it is used in essentially the same way that the hardware associative memory is used in remembering references to segment numbers and page numbers. The table may be of limited size and infrequently referenced entries can be deleted to make way for other entries which are more frequently referenced. Thus, the most referred to local names will be in the table. A future reference to a local name which gets deleted from the table will work correctly but will just take a little longer, as long as it did the first time it was ever referenced.

There is another place where we can use this technique, in search control. Search control can maintain a table whose entries are tree names with their corresponding segment numbers. We can make a significant gain in efficiency by using this table if we change slightly our search algorithm. We change to a recursive type search, i.e., we work from right to left in the tree name rather than from left to right as was previously done. The rule for a recursive type search is: peel off the rightmost component and see if what remains is ROOT, if not, call search control. This is a recursive call to get the

segment number of the prefix which we have retained. The prefix which we have retained is, in fact, the name of the directory in which we expect to find the segment whose name is the rightmost component. Applying this algorithm to our previous example we would make in total three calls to search control. The original call would be a request for the segment number of ROOT.X.Q.Z. The second call is by search control to itself requesting the segment number of ROOT.X.Q. The third call is again a call by search control to itself requesting the segment number of ROOT.X. This is the last call since we are left with ROOT when the rightmost component, X, is removed. The segment number of root is built in. Now we are able to search the root directory for X.

Sequence of calls to and returns from search (for tree name ROOT.X.Q.Z)

	input argument	segment pointer returned
1st (original) call	ROOT.X.Q.Z	-
2d call	ROOT.X.Q	-
3d call	ROOT.X	-
return from 3d	-	X#
return from 2d	-	Q#
return from 1st	-	Z#

Contents of associative table after above sequence of calls

tree name	segment number	
ROOT	ROOT#	← this is a permanent entry
ROOT.X	X#	
ROOT.X.Q	Q#	
ROOT.X.Q.Z	Z#	

Figure 9

The recursion begins unwinding at this point and ultimately we reach the segment number of Z. After this search is performed our table will contain the entries shown in figure 9. If search control consults this table before calling either itself or directory manager, any subsequent calls to search control for the segment number of ROOT.X.Q.Z will produce Z#

immediately. No further calls are required since ROOT.X.Q.Z will be found in the table. In addition, we also gain when search control is called to search for the segment number of any segment in any directory which has previously been referenced. For example, a call to search control for the segment number of ROOT.X.Q.W would proceed as follows. ROOT.X.Q.W would not be found in the table, the rightmost component would be removed and ROOT.X.Q. would be found in the table. Thus, we obtain the segment number of the directory in which to search for W immediately without any further calls.

The use of software local associative memories is fairly common in a large system such as our model. It should be emphasized that the use of local associative memories adds no new facilities. They are used strictly for the purpose of greater efficiency. Often the use of such a feature is not explicitly realized although it is used. Hence, the mechanism of the software associative memory is confused with the concepts which are being implemented. Sometimes it is not clearly recognized, even by the designer of the system, that the capability of the system would not be restricted even if a much simpler but less efficient algorithm were used. We are not impling that systems should be implemented without the use of such techniques. However, it is our feeling that systems should initially be designed without the use of such techniques so that the designers can focus clearly on the essential facilities to be provided by the system. Questions of local optimization should be treated later when they do not confuse the fundamental structure or fundamental problems that the system has to deal with.

7. Controlled Sharing of Segments

In stating the objectives of our model we saw that they implied the necessity of being able to share information which is deposited in the system. In this section we explore the implications that this ability to share information has on the structure and working of the file system. We further examine how sharing of information can be controlled so that the privacy of each user can be guaranteed.

7.1. Basic Requirements

First we state two fundamental requirements in order for any control of access to information to be effective.

a. Authentication of user identity. It is absolutely critical that the system be able to authenticate the identity of any user who approaches the system. If one can approach the system, pose as another user, and be accepted by the system as that other user, then the system can not enforce any effective control on access to information in the system. An effective method of authenticating the identity of the user, which is secure enough in most cases, is for each user to be assigned a password which he and he alone knows. If he is unable to supply the password the system will not accept him as who he says he is.

b. Restriction of the use of hardware instructions. It is absolutely necessary that certain of the hardware instructions be prevented from being executed in user programs. The hardware must have the equivalent of a system mode and a user mode. In system mode all instructions may be executed. In user mode only a subset may be executed. If a user is able execute all of the instructions then he can get any of the information stored in the system simply by programming the proper sequence of input/output commands for the disk. These instructions must be blocked from execution in user mode. In system mode all instructions are executable because the system must be able to read and write information belonging to the user on his behalf. Additionally, there must be some hardware partitioning of memory so that the user is unable to modify the system programs themselves. Otherwise he could make changes in them which would allow him to circumvent all of the other protection features in the system.

7.2. Software Considerations

For the purpose of controlled sharing we are going to let the owner of a segment designate who may share this segment. In addition, he specifies what kind of access is permitted for each of the users who may share it. (Note: If the owner himself is included in this access specification he has some self protection.) The kinds of access permitted and specifiable are: read, write, and execute, or combinations thereof.

Additional information must be added to each directory entry. The additional information required is:

a. Identification of the owner.

b. An access control list (ACL).

The access control list is a list of pairs of the form (user_id, access). The user id is either the name of an individual or the designation of some group (which may be everyone). The access is either read, write, execute, or a combination thereof.

Each process has a small private data base which contains information which is peculiar to that particular process. The information includes such things as: who owns this process, the time logged in, the account number which charges are to be charged against, etc. In a sense, these are the machine conditions of the pseudo-processor. Whenever the file system is asked to reference a file the access control list in the directory entry is checked. If the owner of the process which is currently executing, and hence making the request, is not on the list, or is not included in one of the group designations on the list, then the file system will refuse to manufacture a descriptor for the segment.

7.3. Hardware Considerations

This leads us to the point where it should be clear that in order to inforce access control some hardware help is required. First, since the hardware cannot reference any information except by a segment addressing, if there is no descriptor for a segment in the descriptor table a process is unable to make any reference to the segment. In order to inforce the type of access, once it is known that access is permissable, we need some additional information in the descriptor. The additional information needed is the kind of access (i.e., read, write, or execute) which the referee may have to the segment. By using different descriptors, different users may access the same segment, but with different privileges. In the diagram the owner has read and write access and the sharer has only read access. The absolute address in the two different descriptors points to the same place, namely the page table for the segment. Notice also that we have added a length field in the descriptor so that the hardware may prevent references to non-existent pages. To summarize, each process then has its own private

descriptor table, which is pointed to by the descriptor base register. As a result of this each process can have different descriptors, each process may have different access to the same segment, and each process may have different segment numbers for the same segment (which aids allocation of segment numbers). Finally, a process may not access any information other than that which is reachable through descriptors and the descriptor table, even when using machine language instructions.

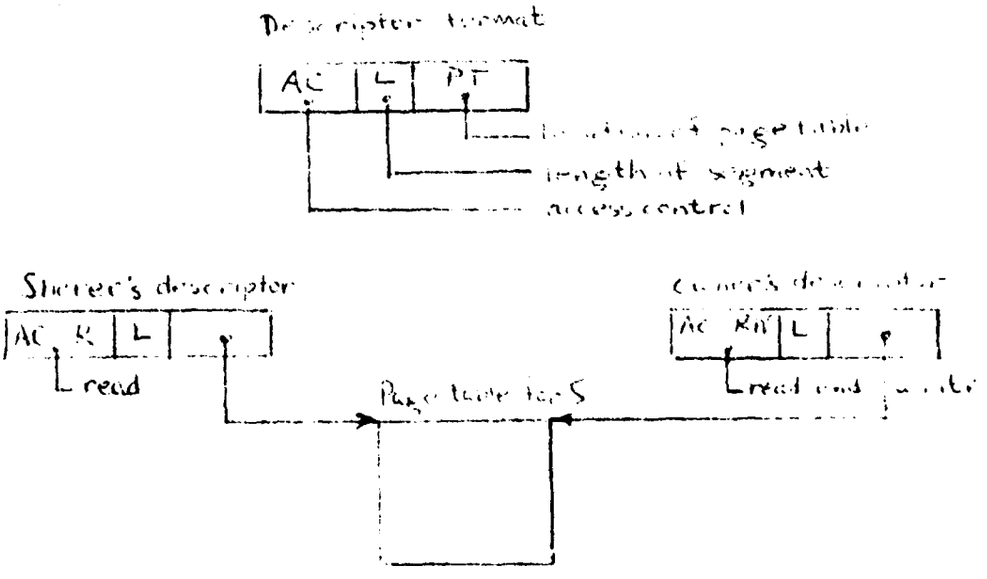

Figure 10

7.4. Immediacy of Sharing

There are two degrees of immediacy of sharing. When designing a system such as our model we must decide which of the two degrees will be permitted. The first degree is to interlock the entire segment while any user is modifying it for the entire duration of his modifications. Anyone attempting to read the segment will have to wait until the segment is released by the writer. In order to capture a segment for writing the writer has to wait until all users who are reading the segment have released it. The second degree is one in

which interlocking is left upt to the users. This degree of immediacy is sufficiently broad so that the previous degree is a special case of it. If the previous choice is made, then one does not have the alternative to implement, as part of the system, the second degree of immediacy. We choose, as was implied above, the second degree. This is made possible because of the ability to have multiple descriptors for the same segment. The system will of course supply some utility routines to assist the user in managing the interlocking of portions of data segments. However, no presumption is made by the system that one particular interlocking discipline is better than another.

7.5. Sharing of Procedures

The sharing of procedures has further implications which we now explore. The ability to have pure procedures (i.e., ones which do not modify themselves), especially when they are system procedures or popular commands, gives a rather significant payoff in terms of space saving. It is a fact, however, that most procedures need some private data in order to function properly. One common example of private data are links to external segments. Recall that the environment of a process provides a private data area for procedures. It is to be expected that if several processes are sharing the same procedure segment the private data required to make that procedure segment function properly for each process will be different for each process. Hence, one implication of the sharing of procedures is that the private data area will have to be duplicated for each distinct process sharing the segment. The private data area in the environment of the process is then private to the process, as is the stack and the pseudo-processor machine conditions which were mentioned earlier.

The information needed to make up the links which are part of the private data area are an appendage to the procedure and we wish to have only one copy of the procedure. Hence, the linker is faced with an additional chore. When establishing a link to a segment for a first time the linker must make a copy of the appendage, which is a template of the procedure's private data area, into the private data area of the process. The private data area of the process is sometimes called the linkage segment. The standard calling sequence used to call procedures is designed to maintain a hardware base register which

always points to the appropriate private data area for each procedure while it is executing. In addition, the standard calling sequence also maintains a base register pointing to the stack area available to the executing procedure.

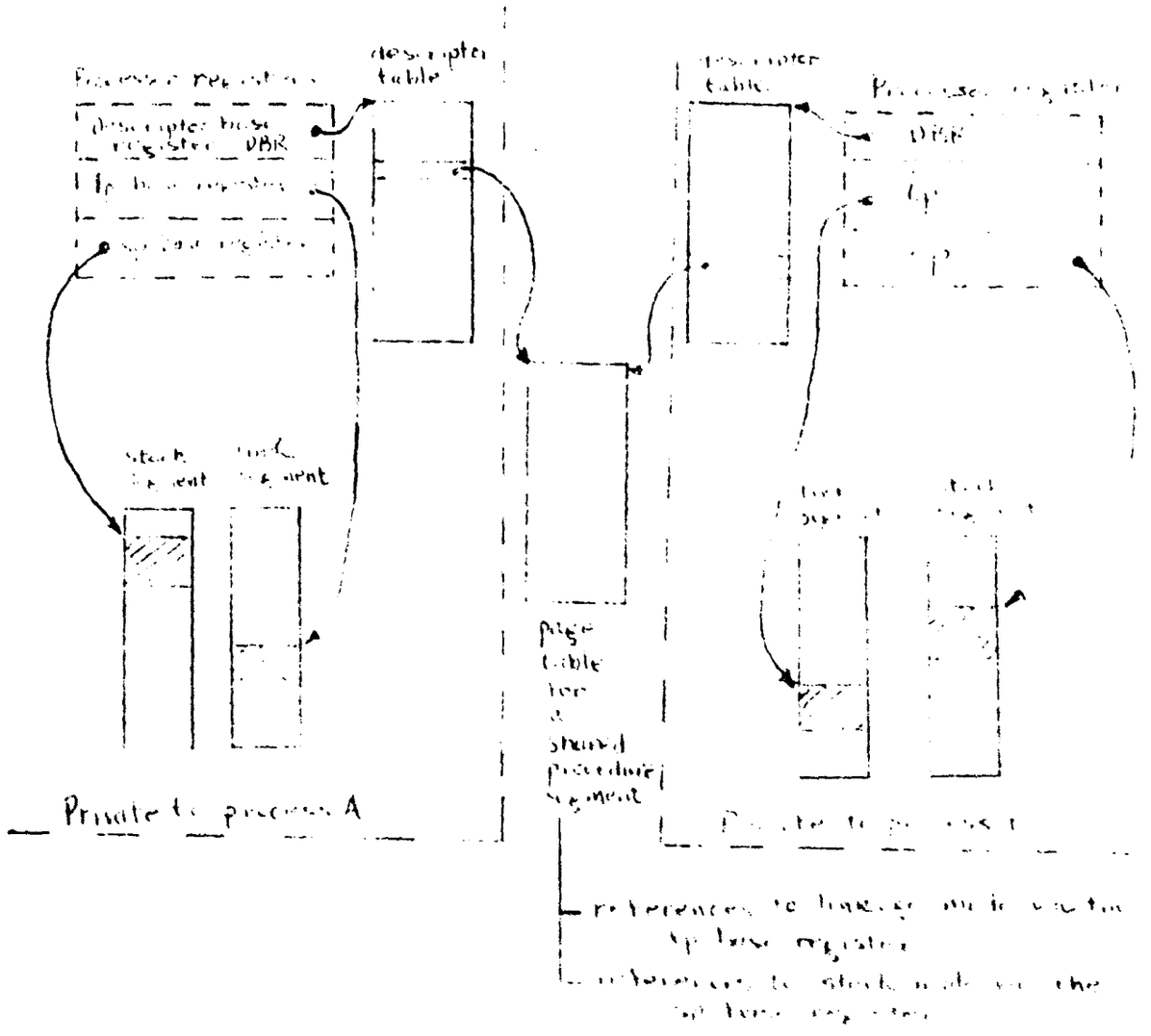

Figure 11

8. Paging

So far we have assumed essentially unlimited core memory. We have proceeded a long way in our discussion of the problems of a file system: management of the hierarchy, mapping between the hardware segment address space and the hierarchy name space, and several other topics. We now discard the assumption that memory is essentially unlimited and examine the consequences of core memory being quite small. Remember that information must be in core memory in order for the processor to use it. Thus, the file system, as we view it, must be concerned not only with the mapping between the hierarchy name space and the segment address space, it must also be concerned with the movement of information between the file memories, such as, disk, drum, etc., and core memory where it can be directly referenced.

One simple, straightforward solution to the problem of a limited amount of core memory is to restrict the user address space to be only that size and force the user to reuse the addresses. This forces him to program all of the memory allocation. This has been the common solution in the past. Systems which choose this solution usually have a feature called chaining or overlays. All these facilities do is to help the user manage core memory, especially the interchange of programs between the core memory and the disk or drum. These facilities typically provide no help in the management of the program's address space. We reject this solution for the following reasons:

1. The user has to be concerend with the details of address space management. Our philosophy in building ever more complicated systems is to provide more and more services for the user, making the system more convenient for him to use and relieving him of concern for problems which are not basically part of the problem he is seeking to solve.

2. The user will probably not do an efficient job of memory allocation. In fact, in a time-sharing, interactive environment the user cannot do a good job. Given the frequency of interaction, the resultant rapid movement of user information in and out of core memory is required in order to achieve a suitable response time. In this situation the user is unable to predict when or how to move his own information around. The system must work with a global view of all activity in the system.

3. All users can benefit from the sophisticated memory management algorithm which the system designers are able to implement. Further, they need not pay the price of having copies of their own management algorithms which, although they may

be equally sophisticated, will not be significantly better.

4. If the system is responsible for the management of all memory, then when the size of memory, the type of memory, or the file memory devices are changed or modified in such a way that a change in tactics is required, only one program, namely the system memory management program, need be changed. The user programs will continue to execute unaware of the change having taken place.

8.1 Models of Program Behavior

We are going to discuss some memory management techniques and some of the considerations that go into the design of a memory management algorithm. First, we look at two models for program behavior. In order to design a reasonable memory management algorithm we must have some model, some idea of the behavior of programs in general. For the purpose of discussing the two models, let us suppose that (N-n) words of bulk (B) memory are available with access time (to an individual word) T and n words of local (L) memory are available with access time t, with t<<T and n<<N.

Our first model is one of completely random access. We assume that the probability that any given word will be referenced next is the same for every word in memory. We are interested in two probabilities: the probability that the next reference will be in local memory, which is $P_L = \frac{n}{N}$ and the probability that the reference will be in bulk memory, which is $P_B = 1 - \frac{n}{N}$. We see that the average time for a reference is $T_{ref} = \frac{n}{N} t + (1 - \frac{n}{N}) T$. Figure 12 shows T_{ref} plotted as a function of the size of the local memory while the total memory size is kept fixed.

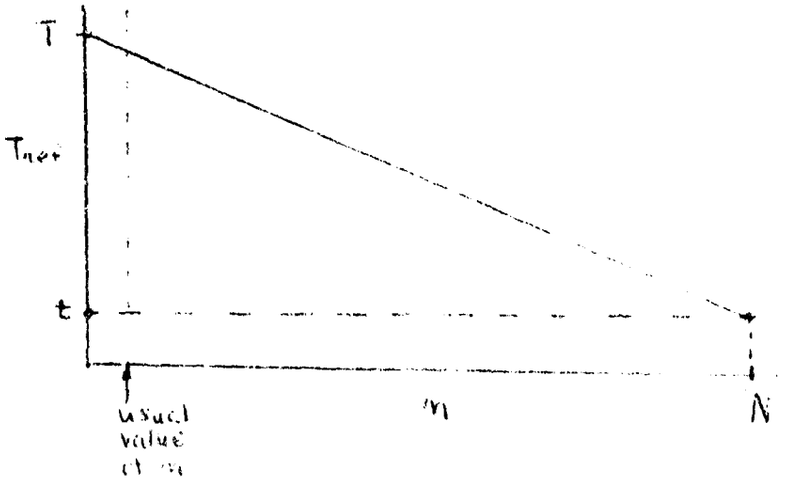

Figure 12

We see that T_{ref} is largest when all of the memory is bulk memory and smallest when all of the memory is local memory. Let us look at the case for a typical system existing today. Common sizes are, n = 128K (approximately 10^5) and N = 10 million (approximately 10^7). Common access times are t = 1 microsecond (approximately 10^{-6}) and T = 10 milliseconds (approximately 10^{-2}). Using these values we see,

$$T_{ref} = \frac{10^5}{10^7} 10^{-6} + (1 - \frac{10^5}{10^7}) 10^{-2} = 9.9 \text{ milliseconds}$$

a rather high average reference time.

In this model there is no correlation between successive references. In general this is never true for an actual program, hence, this model represents the worst possible case. Any program which executes very long has loops in the program. Furthermore, instructions usually follow in sequence, thus there is a high probability that, after having referenced one word in a procedure, the word in the next sequential location will be referenced next. A loop tends to concentrate the references in a small area of the program for some substantial period of time. Another property is that many data manipulations deal with contiguous blocks of data, e.g., matrix and vector operations, table searching, and text editing. This again increases the probability that the next reference will be to a word which is contiguous with the previously referenced location. Again the references over a period of time will be concentrated in some area. Thus, both procedure and data tend to concentrate references in some small area or areas.

We state this fact of program behavior as the principle of locality: The next reference is most likely to be to a word near one which has recently been referenced. Let Δt be some small time interval. Figure 13 shows the frequency of reference plotted against memory location. For the random model we see the frequency of reference is evenly distributed across the entire memory with very little variation. The actual situation is more like the second graph in which there are a number of peaks which are areas of frequent reference.

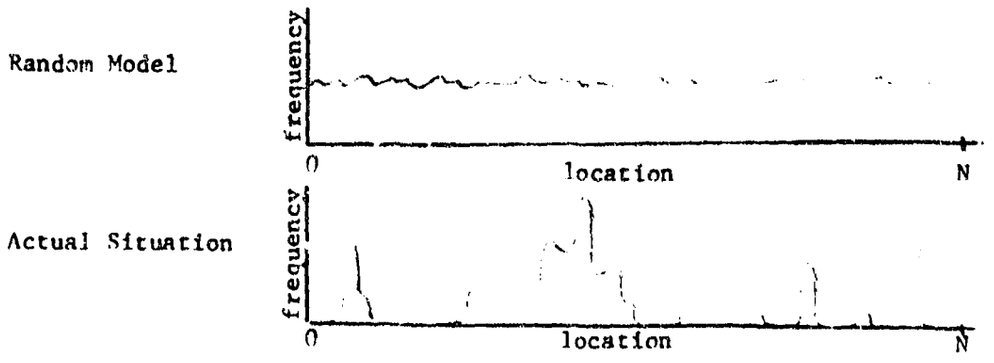

Figure 13

In the second model we collect all of the words with a high frequency of reference into the local memory. We further assume that k references to local memory are always followed by one reference to bulk memory. Our formula for the average reference time is

$$T_{ref} = \frac{k\,t + T}{k + 1}$$

In figure 14 we have plotted T_{ref} against the size of the local memory for $\Delta t = k+1$. We notice that the curve for $n = 0$ starts at T and slopes down quickly to an average reference time of just slightly longer than t when $M = k$. Increasing the size of the local memory beyond k decreases the average reference time very little. We conclude from this that if the local working memory is large enough to hold the frequently referenced information then little gain in overall average reference time is achieved by making substantially more memory available. We define the working set to be the collection of words which are referenced during a time period Δt. Thus, we see the average reference time is quite dependent upon the working set being small enough to fit into local memory or alternatively, the local memory being large enough to hold the working set. Of course the working set changes with time as the reference pattern changes. Thus, in order for our storage management to be effective we must design the algorithms so they adopt to changes in the working set insuring that on the average the working set is in local memory. This is the problem in paging; namely, when are pages brought into local memory and what pages do we push out to make room for them. Given a particular model of program behavior it may be possible to find an optimal algorithm for making these decisions based on that model. However, at present no single model seems to be good enough.

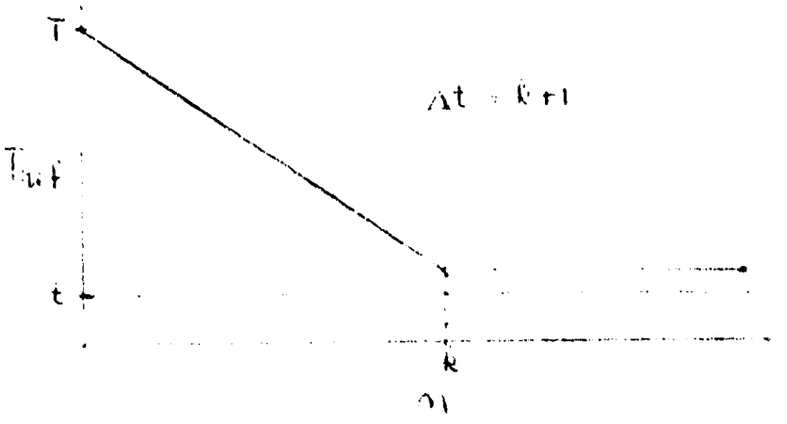

Figure 14

8.2 Paging Techniques

One technique which does work rather well in practice is demand paging. The system assumes that the working set is contained in those pages which were most recently referenced by the process. Hence, whenever a new page, one that is not currently in memory, is referenced it is added to the working set: that is, it is paged into local memory. No attempt is made to page in any page before it is referenced; this is because there is no reasonable way for the system to predict which pages will be referenced in the near future if they have not been referenced in the near past.

In order to make this kind of prediction the system would have to know more about the structure of the program, particularly with respect to the flow of control through the program. It might be possible to make this kind of prediction in the future with some assistance from the language processors. If a language processor were able to construct a skeleton of the control flow and include with it information about the frequencies of loops and the transitions from one part of the program to another, it is conceivable that this would be sufficient for making valid predictions. However, flow analysis of existing languages is at best difficult. Thus, it seems unwise to build a system around this predictive paging decision without a great deal more study of program behavior models.

Demand paging is a page-in technique. As a companion of any page-in algorithm there must be an algorithm for replacement (or page-out). A replacement algorithm must decide which page to put out when more space is needed in local memory. Using the working set concept, we assume that page which was referenced longest ago is no longer part of the working set. It is the first candidate for paging out. Again without more information about the program's behavior this is probably the best that can be done.

Within the framework of a system with many interactive processes, other considerations for paging in and paging out may actually be more significant than the question of which page to page in and which page to page out within a single process during its execution. In our model with many users and many processes simultaneously active, it is expected that control will be switched rather rapidly from one process to another; hence, the processes are in competition for local memory, each needing enough local memory to store its working set if it is going to execute rapidly enough to respond in a reasonable time. The local memory must be large enough to hold all of the working sets of all of the processes which are currently being executed by any of the processors. Further, it must be able to hold the working sets of all processes which are candidates

for executing next. The local memory should be able to hold the working set of all the processes that are on the ready or running lists. Otherwise the following problem arises.

A process needs a new page. While waiting for the page to arrive it goes blocked. The traffic controller now attempts to execute the next process on the ready list. If this process's working set is not in core, the system will have to page in a page before being able to execute the process. In general, the system will need to page out a page of the process which it just blocked in order to make room for the new page of the next process. While the new process is waiting for a page of the old process to be paged out and its page to come in, the required page of the old process arrives. The old process now resumes execution, but, immediately references the page which just got paged out. To make room for this page, a page of the new process gets paged out. Around and around we go. This is thrashing. It can be avoided only by having enough local memory to hold the working sets of all the processes that are on the ready and running lists.

What about processes that are on the blocked list? There are three considerations here. First, if it's blocked waiting for a page to come in we certainly don't want to put out any of its pages unless we absolutely have to. Second, if the process is blocked waiting for typewriter input we know the typewriter response is relatively slow, in the order of seconds, even if the user at the typewriter console can react very fast. All pages belonging to this process that are currently in local memory are excellent candidates for page-out since they will not be needed for some time. Third, when a process interrupted by the system because its time slice was used up, is restarted its working set is probably the same working set that it had when it was arbitrarily terminated by the system. Hence, some prepaging at this point is reasonable and the working set should be restored to local memory before starting execution of the process.

Two further observations should be made. There are some pages that can not be paged-out at all, e.g., the pages containing the paging procedures themselves and interrupt handlers. Secondly, pages which are unaltered need not be written out. In other words, page-out of a page which has not been modified since it was paged in consists of returning the block to the free storage and marking the page as having been paged out. The next time the page is referenced the page-in consists of rereading the original copy.

8.3 Hardware Considerations

In order to implement the techniques of the last paragraph, we need to expand the page table word to include some additional fields. The page written field is set by the hardware whenever a store operation refers to any word in the page. The page reference bit is set by the hardware whenever a reference of any kind is made to the page. The page missing bit is set by the software to indicate that the page is no longer in core. The field which contains the absolute address of the page when it's in core is used to indicate, indirectly, where the page was stored in secondary memory. Its contents will be used as an index in the file map for the segment. The file map is a table indicating the location of each page in secondary storage.

Any attempt to reference a page when the missing page bit is set causes a fault. The fault handler is the paging program. The paging program, using the fault information finds the information describing the location of the page in secondary storage, finds a free block in core, and sets up a request to read the page into that block. Once the request is started, the process is blocked until the page arrives so that some other process can use the processor. When the process awakens the page will have been read into its assigned block. The pager then puts the absolute address of the block into the page table word and clears the reference and written bits. Control is then returned to the place in the program where the page fault occurred.

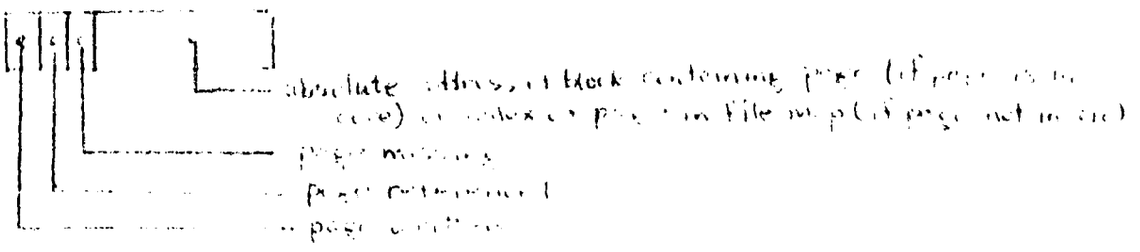

Figure 15

8.4 Memory Swapping

We mention at this point an alternate method of memory management which has been used in several systems in the past and which still receives considerable support: the complete swap. In the complete swap, all of the program and data for a user is swapped out whenever his process stops execution and all of the program and data for the next user is swapped in before he starts execution.

This method enjoys the advantage that it is very simple to implement. On most of the older hardware it is also rather slow, however, some of the modern hardware alleviates this problem with special design which makes swapping very rapid. The method suffers two significant disadvantages:

Much more information is moved than is needed to be moved. With the paging algorithms described above it is clear that a large part of the data and program in a large program complex is not in memory much of the time and is never read into memory except when it is needed. The counter argument in favor of the complete swap is that this movement of redundant information really costs nothing since the entire swapping process is so fast. This may be the case in terms of speed if the hardware file storage devices are sufficiently fast. However, this probably results in higher cost for the hardware. The second disadvantage is that the complete swap method makes flexible sharing impossible or difficult enough so that it is practically impossible. Flexible sharing depends upon the users' physical information being fragmented into small enough entities so that they can be moved around at different times rather than being welded together into a single piece all of which must be moved at the same time. It is clear that flexible sharing may result in many users sharing one segment with small subsets of these users sharing disjoint sets of segments, until the pattern of sharing becomes quite intricate and complex. The only possible way to permit this kind of sharing is to physically fragment the information. The separate segments can then be moved in and out of local memory based on their usage, rather than based on the status of any particular user of a segment.

8.5 Management of Multi-level File Storage

We have been looking so far at the movement between local memory and the first level of secondary storage. As mentioned earlier, secondary storage actually comes in a number of different levels. Local memory is usually core memory. The first level of secondary storage is either drum or large core storage. Since both drum and large core storage are too small to store all of the files in the system additional levels of memory are necessary and must be used by the system for storage of information. The next level is ordinarily disk. The level beyond that would be tape, data cell, or some other similar device.

A decision algorithm is required for the movement of information between each pair of levels. We have looked only at the decision algorithm for the movement of information between core memory and drum. It is not our intent here to

discuss decision algorithms for movement between the other levels. We will simply state a philosophical principle which can be used for guidance in determining the algorithms. The principle is that the oldest information, in terms of last reference, should be stored on the slowest device. So, in a sense, the movement algorithms are attempting to order the information on the various devices in such a way that the information which has been most recently referenced is stored on the fastest access device and information which is the oldest is stored on the slowest device.

9. Address Space Mapping and Paging Combined

Let us look at the complete picture of the management of the hierarchy name space, the management of the segment address space, the mapping between them, and local memory management (paging) which we have just discussed. A directory entry now contains the following information:

(a) name of segment
(b) length of segment
(c) type of segment
(d) access control list
(e) location of segment
(f) date segment created
(g) date segment was last used
(h) date segment was last modified

The location in the directory entry is no longer a simple segment number, since segments may be stored on a number of different devices. If it is stored on disk or tape the location will consist of the device identification and other information which locates the segment on the device. The last three items, which are dates, are new items that are needed in order to implement the movement algorithms between the various levels of storage. Figure 16 shows a flow diagram of the various pieces of the file system which we have discussed. The circles are data bases which the modules reference. The flow diagram represents the complete file system which is active whenever a process is being executed. That portion of the program which implements the movement between disk and tape is a separate part of the system which we will look at in the next section of these notes.

9.2 Segment Address Space Management

Management of the segment address space is somewhat more complicated than the simple picture presented earlier. The argument to the segment address manager is the segment pointer to a directory entry for the segment that is to be activated.

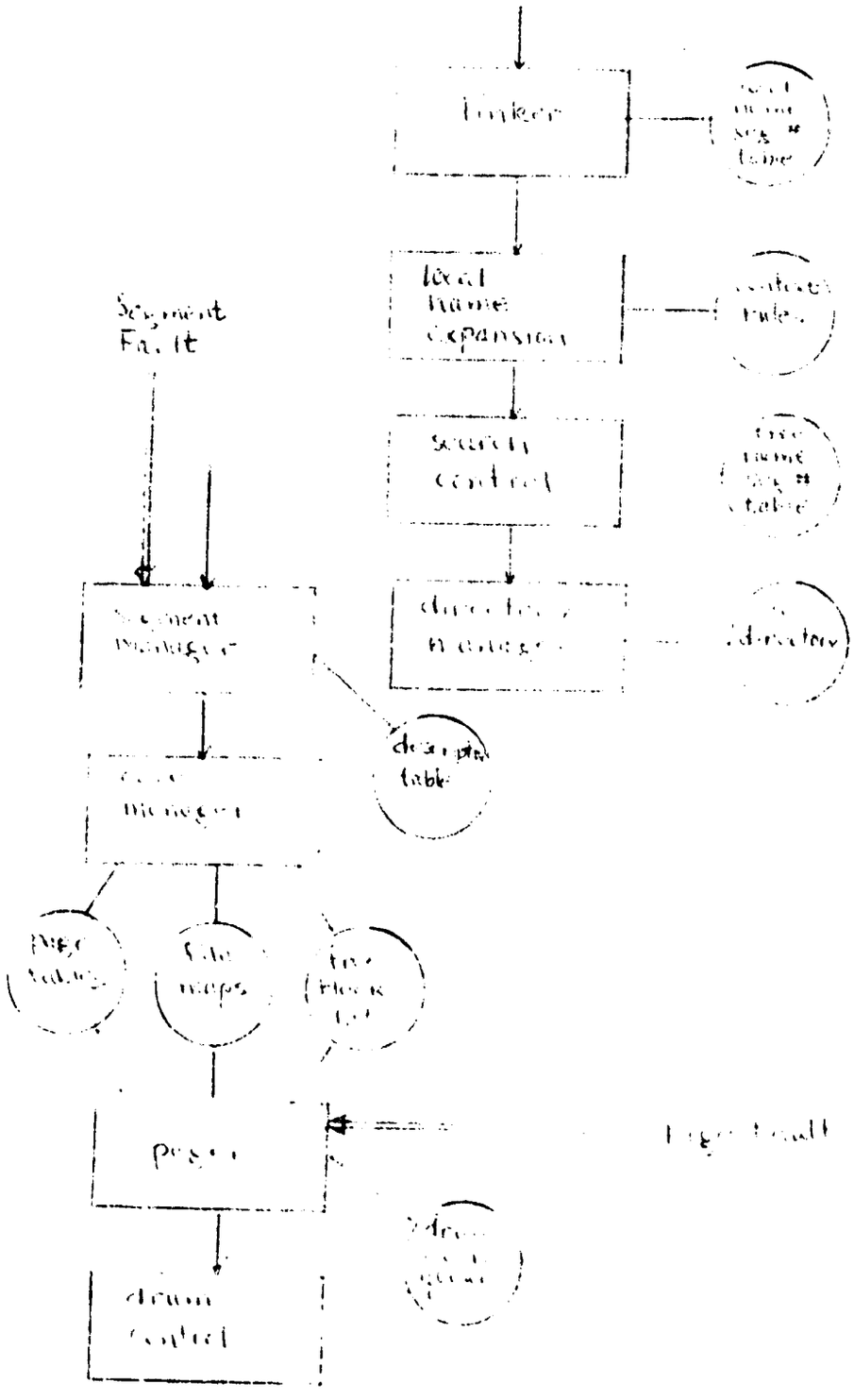

Figure 16

The action of the segment address manager is as follows:

(a) Build a descriptor for the segment using the access control information in the directory entry. Enter the descriptor in the descriptor table, thereby assigning a segment number.

(b) Call the memory manager who sets up the page table for the segment and fills in the page table entries. Initially all the entries in the page table will have the missing page bit set, since none of the pages are presently in core memory.

(c) Build a file map which indicates the location of each page in secondary memory.

(d) Start any information transfers which may be required.

(e) Return the segment number to the caller.

A strategy question arises with regard to the movement of a segment which is stored on the disk into core memory. The particular strategy depends on the specific hardware: the transfer time between disk and core, the transfer time between drum and core, and the number and nature of the transfer paths. In the typical modern computer configuration it is probably best to page a lightly referenced segment directly to and from the disk, while a heavily referenced segment should be paged to and from the drum. It probably is unwise to begin an en masse movement of the segment from the disk when it is first referenced for the same reason that prepaging is ineffective. However, if the segment is stored on tape action to retrieve the segment should be initiated immediately.

One further problem arises. A process may reference enough different segments that local memory gets filled up with page tables. It is clear that we need a way to get rid of page tables, the page tables for segments which have not been referenced in some time. If a segment has not been referenced in a long time all of its pages will have been paged out. Clearly, there is no need for the page table until the segment is referenced again. Hardware assistance is needed. The segment descriptor is extended to include a missing segment bit. When the missing segment bit is set the page table may be discarded. If an attempt is made to reference a segment whose missing segment bit is set, the hardware will generate a missing segment fault. The fault handler for the missing segment fault will reactivate the segment by building a new page table for the segment and a file map.

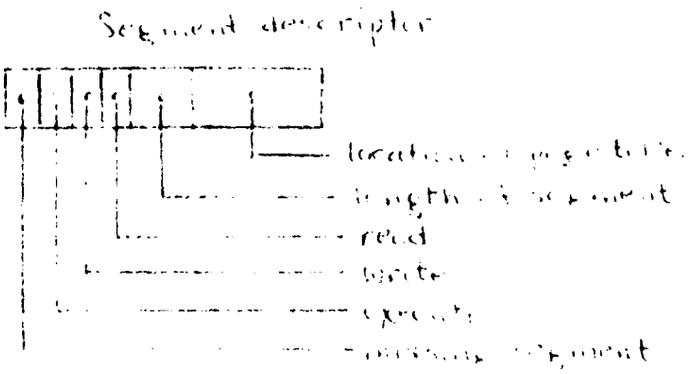

Figure 17

10. Backup and Retrieval

In this section we discuss the following problem. No hardware device and especially no software system is totally and continuously reliable. The question then arises: What do we do when a malfunction occurs? Presumably we wish to pick up the pieces as best we can, restore service as soon as possible, and minimize the amount of lost information. The malfunctions range from complete distruction (e.g., all of the disk platters are scored) to minor destruction (power failure, minor hardware or program bug) where most of the information is intact and only a small amount is lost. To recover from any information loss, backup copies of all the information is required. Lost information is then restored from one of the backup copies.

10.1 Complete Dumping

A straightforward solution to the problem is to dump everything on tape one or more times per day. When trouble occurs everything is reloaded. This solution has a number of disadvantages. It requires an excessive amount of time to dump. In CTSS on the IBM 7094 with more than 30 million words of disk storage used, the required time was 4 to 5 hours to dump. While dumping was taking place essentially nothing else could be done with the computer. An excessive amount of information must be stored in this solution, most of it is redundant. Again in CTSS a complete dump required 6 to 8 reels of tape. Service can not be restored rapidly. The reloading procedure on CTSS took from 5 to 6 hours before service could be restored again. Finally, since the periods between dumps will have to be at least a day if any useful work is going to get done on the computer, the amount of information lost when trouble occurs is not very minimal. Some scheme

is required which avoids storing so much redundant information. Such a scheme will of course be more complicated than the simple straightforward solution posed in this paragraph.

10.2 Incremental Dumping

Three situations may occur which should be distinguished when considering the problem of backup:

(a) The situation is hopeless, the entire contents of the disk must be restored.

(b) Most of the disk is alright and only selective restoration is required.

(c) No restoration at all is required, a few inconsistencies exist which need to be resolved.

We name these three situations respectively, reload, retrieval, and salvage. If we are to avoid saving excessive redundant information then we must dump only information which has been changed recently (since the last copy was saved). Note that some redundancy is desirable; saving two copies is not considered excessive redundancy. It is good insurance against failure to make one of the copies correctly. In addition to backup, which we are discussing, we intend to couple the last stage of multi-level file storage management with the backup mechanism, i.e., old files will be kept only on the backup tapes. In other words, the last level of file storage is tape, the backup copies of the files.

In an attempt to satisfy the objectives of restoring service quickly and minimizing the amount of redundant information stored we will use three different kinds of tapes; incremental dump tapes, system checkpoint tapes, and user checkpoint tapes. The incremental dump tapes are written by a system process (the daemon) which is continually active (although it spends much of its time in a blocked state) dumping files onto the incremental dump tape. All of the segments belonging to a user which were modified during his operating session are copied at the end of the session onto the incremental tape. If a session runs longer than a day then the files modified will be copied onto the incremental tape once a day. In addition, any directory entries which have been modified during the session are also copied onto the incremental tape. Hence the incremental tape contains a copy of all information which is modified during the day. These tapes are created continuously. Figure 18 shows the creation of incremental tapes and system and user checkpoint tapes plotted against time.

Figure 18

Since some parts of the directory hiera[illegible] and some of the supervisor segments may not have been modified for a long time, the reload process would have to scan through a large number of incremental tapes in order to get the system started if this information existed only on the incremental tapes. Hence, we need some sort of checkpoint tapes if we are to restore the system to its operating state in a short amount of time. The system checkpoint tapes will contain all the information needed to get the system operating on-line. This includes the supervisor segments themselves, the accounting allocation records, and the complete hierarchy skeleton (i.e., the contents of all the directory entries). The user checkpoint tapes contain all the user data segments which have been referenced since the last user checkpoint tape was written. Notice on the chart that the system checkpoint tapes are written every few days and the user checkpoint tapes every few weeks. These parameters are adjustable to get the best balance between fast restoration of service and excessive time spent in writing these tapes initially.

All of the tapes are written in the same format. A record on the tape consists of either the tree name of a data segment followed by the data segment contents or the tree name of a directory entry followed by the information in the directory entry.

10.3 Reloading

The reloading procedure which is used in the case of total destruction is as follows. The incremental tapes have been written with a header record which

contains complete instructions for the reloading, including the identification of tape reels to be used by the operators in each of the following steps.

(a) Start with the incremental tape being written at the time of the crash. When this tape is loaded by the reload program it prints out, for the operator's benefit, a list of tape reel identifications in the sequence in which they are to be loaded by the reloader.

(b) All of the incremental tapes written since the latest system checkpoint tapes are reloaded in reverse order, that is, the most recent one first.

(c) The latest system checkpoint tape is now loaded. Note that in the reloading process as each incremental or system checkpoint tape is processed no file is loaded which is already in the hierarchy. This means that the latest copy of each file is the one that is retained, even though there may exist copies of a file on several of the incremental tapes. After this step normal on-line operation begins. The supervisor has been restored, the user account information is present, and the hierarchy is complete. Since the hierarchy is complete, we now have complete information on the location, in the backup system, of all files that have entries in the hierarchy. This means that normal operation can begin and users who attempt to use the system will be given precise status information on any files that have not yet been restored to the disk.

(d) Continue reloading all of the incremental tapes written since the latest user checkpoint tape and before the latest system checkpoint tape.

(e) Load the latest user checkpoint tape.

The reloading process stops at this point. Any file which is not now restored has not been referenced since the next to the last user checkpoint tape was written. This is true because the user checkpoint tape copied all files that had been referenced since the previous user checkpoint tape. Thus, we have restored all segments which have been referred to within the past several weeks. Any segments which have not been restored by this process exist on some incremental tape. The directory entry for the segment has complete location information, including tape reel identification. If it should be referenced by the user the system will automatically retrieve it from the appropriate incremental tape. This retrieval is automatic. The system instructs the operator what tape reel to mount. The system then searches for the segment and reloads it without user intervention.

10.4 Multi-level Storage Management

It was mentioned that the backup would serve as the last stage of the multi-level storage management. When the disk gets too full, we examine the oldest

segments on the disk. After being certain that they have been dumped on incremental tape, their storage is released to the free storage pool. Their directory entries are appropriately marked. Any future reference to one of these segments will automatically cause the segment to be retrieved from the incremental tape, just as it was in the case of the older files which did not get reloaded after a disaster.

10.5 Consolidation of Incremental Tapes

Let us look at the life time of the various tapes which have been created by the backup. Only the latest two or three sets of user and system checkpoint tapes need to be saved. More than one copy needs to be saved only in case a problem arises in attempting to read the tape. Extra copies are insurance. The incremental tapes form the permanent backup copy of segments in the system. These need to be saved as long as they contain any segment which is known to the system, i.e., which has an entry in the hierarchy. As time passes and months turn into years, the number of tapes that need to be saved can get very large. There are two ways to reduce the number of tapes which need to be saved.

The first method is consolidation. When the cost of tape storage approaches the cost of execution of a consolidation program, it makes sense to consolidate the old tapes. The old incremental tapes probably contain very few files which have not been explicitly or implicitly deleted. The consolidation program processes the old incremental tapes and compacts all of the files still known to the system onto a smaller number of new tapes. These have the same format as the incremental tapes, in fact they are indistinguishable from original incremental tapes. The hierarchy entries for these segments then need to be updated to reflect the numbers of the tape reels on which the segments are now stored.

The second way to reduce the number of tapes that need to be saved is to impose some absolute time limit for the retention of unused segments, perhaps a year is a reasonable time. The user would then have to take some explicit action if he wished his old segments to be retained longer by the system.

10.6 Retrieval

The retrieval program is a special program which restores a given segment from the incremental tapes. The retrieval program is used by the file system to retrieve segments when they are referenced by a user after the file has been deleted from the disk due to age. It is also used by the system to repair minor damage done by a malfunction, i.e., when only a few files are destroyed these files are eliminated from disk, then when they are first referenced by the user they

will be restored from the backup.

10.7 Salvage

The salvager is a program which checks for inconsistencies in the hierarchy, in the free storage pool for the various devices, etc. It will attempt to resolve any inconsistencies that it can and will report all inconsistencies to the operator. The basis for this program is redundancy in the structure of the hierarchy which can be used to detect inconsistencies. However, we operate on the following basic principle: It is better to lose information than retain incorrect information. So, unless we have a high degree of confidence, information is considered to be bad and will be restored from the backup.

The following are examples of the types of tests that can be made to check the consistency.

(a) Check the format of the directory entry. Directory entry formats are in general variable length since the access control list is of indeterminate length. The counts of the variable portions of the directory entry should be reasonable.

(b) Some of the contents of the directory entry can be checked. The length of the segment given there should not exceed the user's quota. Dates given in the directory entry should all be sensible, i.e., beyond the date at which the system started operating. The location information of files should be meaningful, i.e., device numbers should in fact be numbers of devices that actually exist. Track numbers should be within the range of track numbers for the given device.

(c) All the file maps can be scanned making a list of the disk tracks which are mentioned in the file map. This list of used disk tracks can be compared with the free storage list maintained for the disk. The two lists should not have any numbers in common and the sum total of the two lists should account for all the tracks on the disk.

(d) A check can be made that no track on the disk is assigned to more than one segment, i.e., appears in more than one file map.

(e) The length of each segment can be checked against the number of tracks which are assigned for that segment in the file map. The length of the segment should be less than or equal to the sum total of the number of tracks. On the other hand the length of the segment should be greater than the number of tracks assigned minus one.

Extra redundancy can be built into the system for the purpose of making the salvager more effective. For example, it may be worth the additional cost

to put forward and backwared pointers in each of the tracks on the disk. While these are not necessary because of the file map, they give independent checks on the organization of the segment on the disk. If the forward pointer, for example, pointed to the track which was recorded in the file map then we have additional support for the consistency of the information. If forward and backward pointers made sense but the file map didn't we might feel it would be safe to use a two out of three vote and fix the file map entry so that it pointed to the track which the forward and backward pointers of surrounding tracks pointed to.

A salvager program was used in the CTSS system and proved to be very successful. In general, there was a need to run the salvager several times a week. In general, use of the salvager enabled the system to recover from numerous minor problems without having to completely restore the disk. The design of the salvager is based very heavily upon the details of the particular implementation and must be worked out in that framework. In designing the system one should keep in mind the value of additional redundancy in terms of making the salvager effective.

11. Input/Output

There are two basically different ways of accessing information: Direct reference (random access) and source/sink (sequential; read takes from a source and write deposits in a sink). File systems have been built using either one or both of these concepts as a basis for referencing files. The most common seems to be some variation of the source/sink concept. We have been calling the direct reference method of accessing, the segment concept. We call the source/sink method the stream concept. We chose the segment concept for the basis of our file system because we feel it is more natural for the user and provides a better model for the problems of address mapping and storage management than does the source/sink idea. The problems of storage management and address mapping have to be faced in the design of a file system no matter which method of accessing is used as a foundation.

In the following paragraphs we will discuss the input/output part of our model system. We do this for two reasons: To exhibit its inherent simplicity and to see that by proper structuring files can easily be treated as sources or sinks, i.e., as I/O devices. An input/output system is quite naturally viewed using the source/sink idea.

The basic entity is a stream, either an input stream or an output stream. Streams have symbolic names. The basic operations on streams are read and write. With the exception of some system defaults, a stream must be explicitly attached

to a device, i.e., attach (name, device-spec). A stream may be detached and re-attached to another device if desired. The attachment is dynamic and takes place during execution whenever the attach entry of the I/O system is called.

Figure 19 shows a block diagram of the I/O system. The attach entry causes the name of the stream and the device specification for the stream to be recorded in the stream-name/device table. In addition, attach decides which device control module (DCM) is appropriate and enters its name in the table. The I/O control program's action for read and write is relatively simple. On a read or write entry the name of the stream to be read or written is looked up in the stream-name/device table, the appropriate device control module is identified and the read or write call is passed on to the appropriate DCM.

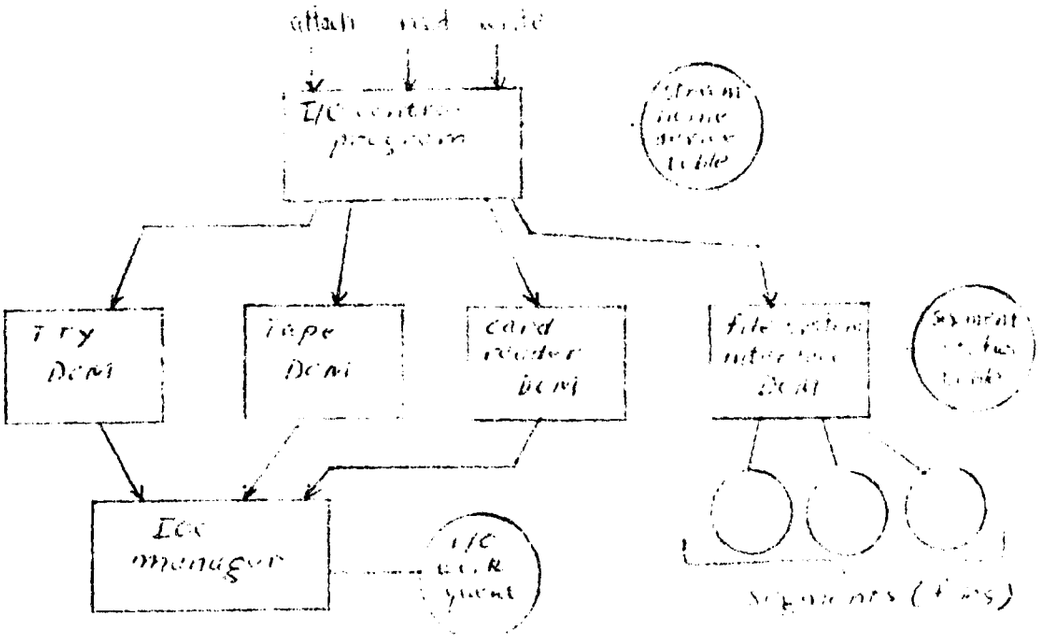

Figure 19

The DCM performs a number of functions. Each DCM has a read entry and a write entry. The DCM converts a device independent request into a device-dependent one. In doing this it must compile a program for the hardware input/output controller (IOC). This program reflects the idiosyncracies of the particular device to which the stream is attached. It may include line controls in the case of remote terminals, select instructions in the case of tapes, and so forth. In addition, the device control module may need to convert the internal character code used by the system into an appropriate character code for the device. Typewriter terminals, for example, come in many different varieties. Virtually every

different variety has a different character code. The device control module after compiling a program for the IOC calls the IOC manager to start the I/O using this IOC program. It is the DCM's responsibility to interact with the IOC manager until this I/O request has been finished. This may require several calls to the IOC manager depending upon the particular format of the programs which the IOC can execute.

The IOC manager is responsible for the overall management of the I/O controller. In general with a large number of different users on the system the IOC manager will have to queue tasks for the various channels of the IOC. The IOC manager is responsible for overall monitoring of the operation of the IOC. This requires answering interrupts, recognizing completion of tasks, and starting new tasks from the queue when channels become free.

The file system interface DCM, functions like any other DCM. However, it does not call the IOC manager. The file system interface DCM is used to make a segment look like an I/O device. The principle data base for the file system interface DCM is a table which contains status information for each segment which is being referred to as a device. When an attach call is made to the I/O control program attaching a stream to a segment, the requested segment is activated. The file system interface DCM maintains in the segment status table an index of the current position in the segment where reading or writing is taking place. Read and write calls are processed by the file system interface DCM and consist of copying the requested information into or out of the segment at the position of the index. After the copy is made the index is updated to the new position in the segment.

BIBLIOGRAPHY

1. B. W. Arden, B. A. Galler, T. C. O'Brien and F. H. Westervelt: Program and Addressing Structure in a Time-Sharing Environment; Jour. ACM, Jan. 1966

2. W. T. Comfort: A computing System Design for User Service; Proc. FJCC, 1965

3. R. C. Daley, P. G. Neumann: A General-Purpose File System for Secondary Storage; Proc. FJCC, 1965

4. R. C. Daley and J. B. Dennis: Virtual Memory, Processes, and Sharing in Multics; Comm. ACM, May 1968

5. C. T. Gibson: Time-Sharing with IBM System/360: Model 67; Proc. SJCC, 1966

6. E. L. Glaser, J. F. Couleur, and G. A. Oliver: System Design of a Computer for Time-Sharing Applications; Proc. FJCC, 1965

7. R. M. Graham: Protection in an Information Processing Utility; Comm. ACM, May 1968

8. M. Schroeder: "Classroom Model of an Information and Computing Service; S. M. Thesis, MIT, Feb. 1969

9. V. A. Vyssotsky, F. J. Corbató, and R. M. Graham: Structure of the Multics Supervisor; Proc. FJCC, 1965

10. P. J. Denning: The Working Set Model for Program Behavior; Comm. ACM, May 1968